BIRT Chart Engine Usage and Integration

org.eclipse.birt.chart.engine usage document

This document provides information on how an external entity may use the BIRT Chart engine in an integration scenario.

Table of Contents

Introduction

The BIRT Chart engine is expected to be used in a variety of applications to provide charting functionality to different applications. To make this easier to do, the following document will list out the integration points and API that the Chart library provides with a view to create, save, load and generate charts in an embedded environment.

References:

Chart Library FAQ:

http://www.eclipse.org/birt/index.php?page=faq/chart.html

Chart Extensions Specification (Engine & UI):

http://www.eclipse.org/birt/

BIRT Newsgroup

http://www.eclipse.org/birt/frameizer.php?page=http://www.eclipse.org/newsportal/thread.php?group=eclipse.birt

Various Standalone Usage Scenarios

This section describes the general usage scenarios for the chart library as a whole.

Creating a chart Programmatically

The chart library operates on the chart in the form of a model. This chart model is made up of classes generated from an XSD model schema using EMF. To create a model would generally involve the following steps:

I. Create an instance of the chart (Chart with axes or Chart without axes)

II. Create additional structural components as necessary (Axes, Series etc.)

III. Set the data definitions for various Series / SeriesDefinition elements

IV. Specify the various attributes and values for chart elements

Example:

An example of creation of a chart programmatically can be seen in the source on CVS. Take a look at **src/org/eclipse/birt/chart/ui/swt/type/BarChart.java** (***getModel***()) in **org.eclipse.birt.chart.ui.swt.extension** plug-in. This method creates a basic BarChart model.

<TBD: Add fully annotated source for creating a chart here.>

Saving a chart

The chart library provides simple API to serialize a chart model to a variety of targets. The API for this are in the **org.eclipse.birt.chart.model.Serializer** interface.

```
    // Write Methods
    /**
     * Write the chart described by the model to the OutputStream provided.
     *
     * @param cModel
     *             The model to be serialized os The OutputStream to which the
model is to be serialized
     */
    public void write(Chart cModel, OutputStream os) throws IOException;

    /**
     * Write the chart described by the model to the location defined by the
URI provided.
     *
     * @param cModel
     *             The model to be serialized uri The URI to which the model is
to be serialized
     */
    public void write(Chart cModel, URI uri) throws IOException;

    /**
     * Write the chart described by the model to a ByteArrayOutputStream.
     *
     * @param cModel
     *             The model to be serialized bStripHeaders Specifies whether
or not the headers are to be removed while
     *             serializing the model
     * @return the ByteArrayOutputStream containing the serialized model
     */
    public ByteArrayOutputStream asXml(Chart cModel, boolean bStripHeaders)
throws IOException;
```

Table 1: API to serialize a chart model

Example:

```
// The following block of code uses the above API to save a chart model to a
// disk file


    String sChartFile = "C:\\Test\\mychart.chart";


    // Create and setup local ResourceSet

    ResourceSet rsChart = new ResourceSetImpl();


    // Create resources to represent the disk files to be used to store the

    // models

    URI URIChart = null;

    Resource rChart = null;


    // Register factories to load & save models in diff. formats with the

    // ResourceSet

rsChart.getResourceFactoryRegistry().getExtensionToFactoryMap().put("chart",
new ModelResourceFactoryImpl());


    URIChart = URI.createFileURI(new File(sChartFile).getAbsolutePath());


    // Get or create the chart instance to be saved

    Chart chart = createChart ();


    // Get the implementation of Serializer

    Serializer s = SerializerImpl.instance();

    try

    {

        // Save the chart to file

        s.write(barChart, URIChart);

    }

    catch (IOException e )

    {

        e.printStackTrace();

    }
```

Table 2: Sample code to save a chart to disk

Loading a saved chart

Just as it provides API to save a chart to disk, the chart library also provides the following API to load a chart from a disk file. These API are defined in the **org.eclipse.birt.chart.model.Serializer** interface.

```
    // Read Methods
    /**
     * Reads the chart model from the given InputStream
     *
     * @return chart model read from the stream
     */
    public Chart read(InputStream is) throws IOException;

    /**
     * Reads the chart model from the location defined by the URI provided.
     *
     * @param uri
     *            URI of the location holding the chart model
     * @return chart model read from the source
     * @throws IOException
     */
    public Chart read(URI uri) throws IOException;

    /**
     * Reads the chart model from the ByteArrayInputStream.
     *
     * @param byaIS
     *            The ByteArrayInputStream holding the chart model
     * @param bStripHeaders
     *            Specifies whether or not the headers were removed when the
chart model was saved
     * @return chart model read from the stream
     * @throws IOException
     */
    public Chart fromXml(ByteArrayInputStream byaIS, boolean bStripHeaders)
throws IOException;
```

Table 3: API to deserialize a chart

Example:

```
// The following block of code uses the above API to save a chart model to a
// disk file


    String sChartFile = "C:\\Test\\mychart.chart";


    // Create resources to represent the disk files to be used to load the
    // model
    URI URIChart = null;

    URIChart = URI.createFileURI(new File(sChartFile).getAbsolutePath());

    Chart chart = null;

    // Get the implementation of Serializer
    Serializer s = SerializerImpl.instance();
    try
    {
        // Load the chart from file
        chart = s.read(URIChart);
    }
    catch (IOException e )
    {
        e.printStackTrace();
    }
```

Table 4: Sample code to load a chart from disk

Generating a chart

Once a chart is created, it needs to be generated and rendered. The chart engine provides the **org.eclipse.birt.chart.engine.factory.Generator** class for this purpose. This class provides the methods to generate a chart to a specified device.

```
// public methods in Generator
    /**
     * Returns a singleton instance of the chart generator.
     *
     * @return A singleton instance for the chart generator.
     */
public static synchronized final Generator instance()    { }

/**
  * Builds and computes preferred sizes of various chart components offscreen
  * using the provided display server.
  *
  * @param ids           A display server using which the chart may be built.
  * @param cmDesignTime  The design time chart model (bound to a dataset).
  * @param scParent      A parent script handler that may be attached to the
existing chart model script handler.
```

```
  * @param bo           The bounds associated with the chart being built.
  * @param rtc          Encapsulates the runtime environment for the build
process.
  *
  * @return  An instance of a generated chart state that encapsulates built
chart information that may be subsequently rendered.
  *
  * @throws GenerationException
  */
public final GeneratedChartState build(IDisplayServer ids, Chart cmDesignTime,
     Scriptable scParent, Bounds bo, RunTimeContext rtc) throws
GenerationException { }


 /**
  * Performs a minimal rebuild of the chart if non-sizing attributes are
altered or the dataset for any series has changed.
  * However, if sizing attribute changes occur that affects the relative
position of the various chart subcomponents,
  * a re-build is required.
  *
  * @param   gcs   A previously built chart encapsulated in a transient
structure.
  *
  * @throws GenerationException
  */
public final void refresh(GeneratedChartState gcs) throws GenerationException
{ }


 /**
  * Draws a previously built chart using the specified device renderer into a
target output device.
  *
  * @param   idr     A device renderer that determines the target context on
which the chart will be rendered.
  * @param   gcs     A previously built chart that needs to be rendered.
  *
  * @throws GenerationException
  */
    public final void render(IDeviceRenderer idr, GeneratedChartState gcs)
throws RenderingException { }
```

Table 5: Public methods in Generator

Example:

The following code snippet shows how a chart is built and rendered to an **SWT** device

```
// Get an instance of the SWT device renderer
IDeviceRenderer idr = idr = ps.getDevice("dv.SWT");


// Prepate the device for rendering
idr.setProperty(IDeviceRenderer.GRAPHICS_CONTEXT, paintevent.gc);


/* Calculate the bounds within which the chart is to be rendered. NOTE: This
calculation makes use of the display server from the device
(idr.getDisplayServer()). The display server provides display-related
functionality */
Bounds bo = BoundsImpl.create(rectangle.x, rectangle.y, rectangle.width,
rectangle.height);
bo.scale(72d/idr.getDisplayServer().getDpiResolution());


// Get the instance of Generator
Generator gr = Generator.instance();


try
{
     // Generate the chart
     GeneratedChartState gcs = gr.build( idr.getDisplayServer(), cm, null, bo,
null);
     // Render the chart to the device
     gr.render(idr,  gcs);
}
catch (GenerationException gex)
{
     showException(pe.gc, gex);
}
catch (RenderingException rex)
{
     showException(pe.gc, rex);
}
```

Table 6: Sample code showing generation and rendering of a chart to SWT device